J Comput Virol (2010) 6:261–276
DOI 10.1007/s11416-009-0126-4

ORIGINAL PAPER

# Automatic binary deobfuscation

**Yoann Guillot · Alexandre Gazet**

Received: 4 January 2009 / Accepted: 17 July 2009 / Published online: 13 August 2009


**Abstract** This paper gives an overview of our research in the automation of the process of software protection analysis. We will focus more particularly on the problem of obfuscation. Our current approach is based on a local semantic analysis, which aims to rewrite the binary code in a simpler (easier to understand) way. This approach has the advantage of not relying on a manual search for "patterns" of obfuscation. This way of manipulating the code is, at the end, quite similar to the optimising stage of most of compilers. We will exhibit concrete results based on the development of a prototype and its application to a test target. Current limitations and future prospects will be discussed in as well.

As a continuation of our work from last year [1], we focus on the automation of the software protection analysis process. We will focus more particularly on the problem of obfuscation.

This problem is crucial as most malicious binaries (like viruses or trojans) use this kind of protection to slow down their analysis and to make their detection harder. Automation is a key step in order to face the constant growth of the amount of malware, year after year.

Our previous paper was mainly focused on the attack and suppression of protection mechanisms using the Metasm framework. It provides many useful primitives to deal with protected code: control flow graph manipulation, recompilation, filtering processor,… Nevertheless most of these approaches rely on a tedious work of manual identification of the "patterns" used by the protection.

Y. Guillot · A. Gazet (✉)
Sogeti, ESEC, 6-8 rue Duret, 75116 Paris, France
e-mail: alexandre.gazet@sogeti.com; alex@security-labs.org

Y. Guillot
e-mail: yoann.guillot@sogeti.com

We will now present the development of our new methods, relying on a semantic analysis of the binary code to extract a simpler representation. The objective is no longer to seek and destroy known patterns, but to proceed to a complete, on-the-fly, optimised code rewriting.

We will exhibit concrete results obtained by applying these methods to a test target. Then, current limitations and future prospects will be discussed.

## 1 Metasm

Metasm[1] [2] is a free binary manipulation framework. Last year, we used it to solve two important reverse-engineering challenges. Based on these works, a few methods have been integrated into the mainstream code. They allow many recurrent analysis tasks to be simplified.

### 1.1 Disassembler

Metasm can disassemble a binary file from one or many entry point(s). It then follows the execution flow and uses its built-in emulation engine to solve indirect jumps and memory *cross-references* (i.e. which instruction reads or writes at which address(es)). This technique is referred to as *backtracking* inside the framework. This concept is similar to the concept of *slicing* [3–5].

Covering the program's code allow the construction of its execution graph (aka control flow graph). The nods of this graph are basic blocks (atomic sequences of instructions—if we do not consider exception that may be raised).

These nods are organised in two interlaced graphs

[1] http://metasm.cr0.org/.

– function's block graph,
– functions and sub-functions graph.

This graph can be visualised using an interactive graphical interface.

### 1.2 New functionalities

Two main improvements have been made to the framework since our last paper.

The first one is a method allowing the graph to be modified by replacing some of its components. This function, `replace_instrs`, requires three parameters:

1. the address of the first instruction of the first block to replace,
2. the address of the last instruction of the last block to replace,
3. the list of new instructions that will be inserted as a replacement (it may be an empty list).

A new block is then built from the new list of instructions, and inserted into the graph, instead of the previously selected blocks.

The second improvement is a method, `code_binding`, that allows to obtain the semantics of a section of code.

The method takes advantage of the backtracking engine which is at the heart of Metasm's disassembler. The engine is called many times to determine the semantics of the code section. It regroups the effects of:

– registers modifications,
– and memory writings.

For the moment, this analysis is limited to code sections with a simple linear structure (without loop or conditional jump). As we will see later, it is nevertheless at the heart of most of our attacks. As an example, getting the semantics allows us to overcome the level of abstraction provided by a software virtual machine based protection.

Finally, the instrumentation of the disassembly engine has been facilitated by the implementation and export of many *callbacks*, allowing us to take control at different moments and to intercept manipulated data. Here are some of those callbacks:

– when a new instruction is disassembled,
– when a jump is detected (conditional or not),
– when some self-modifying code is detected,
– at the end of the disassembler work.

## 2 Case study: a protection analysis

From now on, we use the generic term *packer* to refer to a software protection applicable to a program (at binary or source level), in order to obfuscate its original form and to slow down a possible attacker/reverse-engineer. "Classic" packers, like `AsProtect`, `PECompact`, are usually well handled by security products, anti-virus software and automatic classification tools for example. Many unpacking techniques have been developed over the last few years:

– Static/dynamic unpackers, most of time based on a deep analysis of how the protection works. The unpacking process can later be automated using a scriptable debugger for example.
– Generic unpackers (using code instrumentation [6], or emulation like `Pandora's Bochs` [7] or `Renovo` [8].)

Such protections mainly rely on concepts like compression and encryption built right up against anti-debugging functions, licence management, etc. The main weakness of this class of protection is that, at some point, the code has to be decompressed/decrypted in memory (at least partially) before being executed by the processor. It is then possible to dump and analyse the code.

Close to classic packers stands another class of protection that takes advantage of the virtualization concept. This class of protection is not vulnerable to the previously mentioned attacks, and few generic analysis techniques have been proposed. A part of our research is dedicated to virology and it happens that we encounter many instances of the same virtualization-based protection. Consequently, we have decided to carry out the analysis of this protection.

By quickly comparing the different instances of the protection, we have discovered that we will manipulate to main concepts:

– *Polymorphism*. This concept came to the forefront in the early 90's, with viral codes as the main application field. The challenge was then to try to defeat the signature based detection algorithms used by anti-virus software. By mutating the code's form, it was possible to circumvent a signature, and make the malware go undetected. In order to do so, one could express the same original code semantics using a different sequence of instructions. As the battle raged on between viral code authors and anti-virus editors, the editors tried to react, using more advanced algorithms to defeat obfuscation techniques. Many more formal works have been published on this subject. As an example, in 2003, Frédéric Perriot proposed an approach based on the use of compiler optimisations to improve polymorphic code detection [9]; other

```
023E6C VM_CALL:
023E6C              push    offset vm_call_1
023E71              jmp     VM_INIT
023E76 ; -------------------------------------------
023E76              push    offset vm_call_2
023E7B              jmp     VM_INIT
023E80 ; -------------------------------------------
```

**Fig. 1** A virtual machine call

works were presented in 2008 by Matt Webster and Grant Malcolm [10]. In the same spirit, one should be aware of Mihai Christodorescu's paper [11]. In these two cases, the main idea is to automate the deobfuscation process in order to improve viral code detection rates.

– *Virtualization*. For recall, in the field of software protection, the term *virtual machine* refers to a software component emulating the behaviour of a processor. This *virtual processor* has its own set of instructions and executes programs specifically compiled into the appropriate bytecode. It amounts to adding a new level of abstraction between the machine code that is seen during the analysis (using a debugger or a disassembler) and its real semantics. For more details on the internal workings of a virtual machine based software protection, the interested reader can refer to our previous paper [1].

The approach we will present here is thought to be didactic. At each step of the analysis, we will point out the difficulties encountered and we solved them.

### 2.1 Discovering the protection's architecture

It takes only a few minutes to discover that a virtual machine is used by the protection. When loaded into a disassembler, many big undefined memory areas appear. Furthermore, many distinctive function calls are used (Fig. 1): the original code has been replaced by an initialisation stub invoking the virtual machine; the address pushed onto the stack is actually the address of the bytecode implementing the protected algorithm.

Two classical distinctive patterns (Fig. 2), quite specific to virtual machine-based software protection: a context (actually the virtual machine's registers), and a table of handlers (for recall we consider that a handler refers to the implementation of a virtual opcode/instruction).

So far, so good; no real difficulties. The code is quite standard.

### 2.2 Optimise to tame the code

We have seen that the protection refers to a table of handlers; the next natural step is to analyse them. This will allow us to identify the instructions set of the virtual processor. One can see an example of a handler in Fig. 3. The first characteristic that one should notice is that the code is splitted into

```
0B800 VM_CONTEXT      dd 0
0B804                 dd 0
0B808                 dd 0
0B80C                 dd 0
0B810                 dd 0
0B814                 dd 0
0B818                 dd 0
0B81C                 dd 0
0B820                 dd 0
0B824                 dd 0
0B828                 dd 0
0B82C                 dd 0
0B830                 dd 0
0B834                 dd 0
0B838                 dd 0
0B83C                 dd 0
0B840                 dd 0
0B844 HANDLER_TABLE   dd offset loc_F000C15
0B844                 dd offset loc_F000F5E
0B844                 dd offset loc_F000D2A
0B844                 dd offset loc_F000F8A
0B844                 dd offset loc_F000DF2
0B844                 dd offset loc_F000CF3
```

**Fig. 2** Virtual machine context

```
metasm disassembler

entrypoint_100bef3fh:
lodsb
push cx
jmp loc_100bb5ceh            ; x:loc_100bb5ceh

// Xrefs: 100bef42h
loc_100bb5ceh:
push eax
mov al, 12h
jmp loc_100bed67h            ; x:loc_100bed67h

// Xrefs: 100bb5d1h
loc_100bed67h:
mov cl, 46h
xor cl, al
pop eax
jmp loc_100bbdfah            ; x:loc_100bbdfah

// Xrefs: 100bed6ch
loc_100bbdfah:
add al, cl
pop cx
jmp loc_100be918h            ; x:loc_100be918h

// Xrefs: 100bbdfeh
loc_100be918h:
sub al, bl
jmp loc_100bbdb8h            ; x:loc_100bbdb8h

// Xrefs: 100be91ah
loc_100bbdb8h:
sub al, 54h
sub esp, 4
mov dword ptr [esp], ebx
mov bh, 11h
push edx
mov dh, 0a5h
jmp loc_100beed2h            ; x:loc_100beed2h
```

**Fig. 3** Standard structure of a handler

many basic blocks, linked by unconditional jumps. This kind of code is sometimes referred to "*spaghetti code*". This technique is actually not very effective: in our previous paper we already developed methods to automatically merge basic blocks when needed and to rebuild the control flow graph.

**Fig. 4** Propagation of value 12h through *al*

```
1  mov al, 12h
2  mov cl, 46h
3  xor cl, al
```

```
1  mov al, 12h
2  mov cl, 46h
3  xor cl, 12h
```

**Fig. 5** *cl* Register assignment simplification

```
1  mov al, 12h
2  mov cl, 46h
3  xor cl, 12h
```

```
1  mov al, 12h
2  mov cl, 54h
```

**Fig. 6** Reduction of the *add* computation

```
1  add al, -7fh
2  add al, bl
3  add al, -70h
```

```
1  add al, 11h
2  add al, bl
```

One can also notice that most of the basic blocks imply many basic arithmetic operations, and make excessive use of stack operations. This behaviour clearly stems from an obfuscation process. We need to clean the code in order to be able to analyze it effectively.

The difficulties can now be expressed like this: how can we get rid of the obfuscation with the minimal amount of manual analysis? Our answer was to use compiler optimisation techniques. An optimisation is a code transformation for which many contradictory objectives may be sought-after: speed of execution, final size of code, etc. Our optimisation process has for only objective (our optimisation criteria) to reduce the code to a minimal, concise form. We are not at all preoccupied with performance or size, even if as a side effect of our optimisation process they will also be dramatically improved.

One of the most surprising point about the optimisation techniques we used is their simplicity. From an algorithmic point of view, these techniques are quite affordable and quite effective. For this step, we draw our inspiration from works proposed for an equivalent target [12]. Nevertheless, we did not adopt the same angle of attack, namely working on a textual representation of the code (using a lexer and a parser). Indeed, we already have a representation of all the disassembled instructions: we can directly manipulate Metasm's instruction objects on the fly. Our methods will be performed at the assembly level, inside the basic blocks of the control flow graph.

Here are some of the techniques we have implemented in our optimisation engine:

1. *Peephole optimisation.* It amounts to replacing a known pattern (for example a sequence of instructions) by a simpler form. This technique is, from our point of view, the least interesting because it relies on a manual discovery of those patterns. Nevertheless, for certain precise patterns, it may allow us to avoid using too complex techniques.
2. *Constant propagation.* The basic idea is to propagate the known value of a variable in the expressions using it (Fig. 4).
   The propagated value is `12h`. It can be found at line 1, with register *al* being assigned. On line 3, *al*, which has not been modified since, is then replaced by its numerical value.
3. *Constant folding.* The initial value of a variable is simplified by statically solving some superfluous arithmetic operations (Fig. 5).
   At line 2, *cl* register is assigned with the value `46h`; then at line 3, a `xor` operation with a constant will modify the value contained in `cl` register. We simplify this basic piece of code using a direct assignment of the `cl` register by the result of the operation `46h xor 12h = 54h`. Finally, line 3 is removed from the control flow.
4. *Operation folding.* Once again, a computation is simplified statically, but we do not compute a final result to assign to a variable.
   In this example (Fig. 6), two `add` instructions `add al, -7fh` and `add al, -70h` are joined into a single one. The resulting instruction can be expressed as `add al, (-7fh + -70h)`, that is `add al, 11h`.
   Furthermore, our optimisation engine handles the commutativity of operators, which allow us here to freely reorder a sequence of e.g. `add` instructions, in the most useful way.
5. *Stack optimisation* We did a very trivial implementation of this technique. There are two use cases:
   – A useless *push-pop* couple. For example a register is pushed on the stack and popped without being read or written.
   – An element `a` (for example register `eax`) is pushed onto the stack and then popped into an element `b` for (for example register `ebx`). If possible this

**Fig. 7** Useless push-pop couple

```
push ebx
sub al, 56h
pop ebx
```

```
sub al, 56h
```

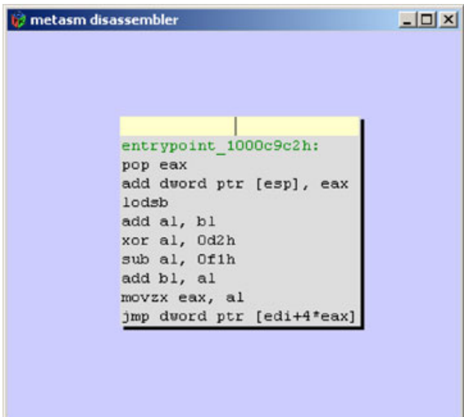


**Fig. 8** Optimised handler

```
lodsb
add al, bl
xor al, 0d2h
sub al, 0f1h
add bl, al
movzx eax, al
jmp dword ptr [edi+4*eax]
```

**Fig. 9** Decryption of the next handler's index

```
pop eax
add dword ptr [esp], eax
```

**Fig. 10** Optimised handler's code

push-pop couple is transformed into the clearer mov b, a instruction.

According to Fig. 7, it is possible to clean the couple of instruction push ebx - pop ebx as ebx register is not modified (no write access) between the two considered instructions.

These different methods are integrated into an iterative process: while at least one of the methods manages to optimise a piece of code, the process is called once again. Despite their simplicity, the first results were beyond our expectations.

The result is quite satisfying: the code of the handler has been drastically reduced. Most of the handlers were initially composed of 100 to 200 instructions and split into a great number of basic blocks. The semantically equivalent optimized code is reduced to at most 10 instructions, all of them inside a single basic block. Actually, a few handlers (less than five), due to their function, are more complex and are still composed of a small number of basic blocks.

We see now that all the handlers share a small final sequence of instructions. This actually is a kind of control stub (Fig. 9), which computes the address of the next handler to execute. This computation is the decryption of the virtual execution flow pointer using a key stored in the ebx register, while the bytecode instruction pointer is located in the esi register.

As a conclusion, we can say that the semantics of a handler only rely on a small number of instructions (Fig. 10).

From a "defensive" point of view, using obfuscation, and by extension, polymorphism, is quite interesting. It can be considered on two different levels:

– locally: it increases the complexity of each of the handlers and raises the amount of work needed to defeat the protection.
– globally: each generated instance of the virtual machine is different from the next (the mutated code of each handler will differ). Thus, an attacker with ineffective tools has to reanalyse each new instance from scratch.

From an "offensive" point of view, the obfuscation engine is by far too weak. Rebuilding instruction through "spaghetti code" is not a difficulty. Even if we work at a very low level of abstraction (inside basic blocks), results are quite self-explanatory. The optimisation engine produces a very clean code and manual analysis has been reduced to a minimum. The module progressively rewrites the code.

Each of the optimisation methods is a rewriting rule, possibly associated with one or more condition. Each of the transformations has to be semantically correct: the optimised code should compute the same function as the original, obfuscated code. Finally, one has to ensure that this engine, or rewriting system, actually halts.

## 2.3 Handler analysis

The previous step allows us to get a clean, optimised code for each of the handlers. Even if the result is very positive, we still are far from our objective. As noted earlier, a method has been added Metasm, which allow the semantics of a section of code to be computed. Thus, we'll apply it on every handler.

```
pop eax
add dword ptr [esp], eax
```

```
dword ptr [esp+4] := dword ptr [esp+4]+dword ptr [esp]
eax := dword ptr [esp]&0ffffffffh
esp := (esp+4)&0ffffffffh
```

```
vmctx = {
  :eax => :eax,
  :ecx => 1,
  :edx => :edx,
  :ebx => @key,
  :esp => :esp,
  :ebp => :ebp,
  :esi => @virt_addr,
}
```

**Fig. 11** Declaration of a symbolic context

We immediately obtain the semantics of the handler. From now on, this set of symbolic expressions will be referred to as the *binding* of the handler.

## 2.4 Symbolic execution

The previous steps of the analysis are totally automated. When our tool faces an unknown handler, it is disassembled, optimised, and finally its binding is extracted. This work is a bit time-consuming, that's why all of this information is stored in a cache: a file containing the description (optimised assembly code and semantics) of each hander is progressively updated during the analysis.

What we get here is actually the whole description of the bytecode interpreter used by the virtual machine.

This result is fundamental. From a theoretical point of view, given two languages $L_a$ et $L_b$, it is possible to find a compiler of $L_b$ in $L_a$, if we know an interpreter of $L_b$ written in $L_a$. This theorem is known as the *second Futamura's projection* [13]. We will see in the next steps how to practically translate this theoretical result.

We have seen that each handler executes a small decryption stub to compute the next handler's index, and then gives it control. The index is stored encrypted, in a cipher feedback mode in the bytecode. In order to be able to trace the instruction flow, one has to know both the value of the key (updated at each round) and the current bytecode instruction pointer. This behaviour looks like the `T2`'07 challenge that we solved last year. Once again, we will tackle the problem using a form of symbolic execution.

We declare a symbolic context (Fig. 11): it is a partial representation of the host processor, namely a standard `Ia32` architecture. On one hand, the symbolic execution is only done on interesting elements; non significant registers are simply not taken into consideration in the representation. On the other hand, many registers are assigned with essential numeric values like *key* and *virt_addr*, which are parameters for the decryption of the bytecode (respectively the initial value of the decryption key and the initial value of bytecode pointer). Other registers are assigned with a symbolic value like *:eax* (the two points are a distinctive character to refer to a symbol in the Ruby language).

```
eax := 2eh
ebp := ebp
ebx := 10016743h
ecx := 1
edx := edx
esi := 10016716h
esp := esp
```

**Fig. 12** Current context

```
lodsb
sub al, bl
sub al, 63h
add bl, al
movzx eax, al
lea eax, dword ptr [edi+4*eax]
push eax
```

**Fig. 13** Optimised handler

We have already obtained the binding of each of the handlers. The next essential step is the "contextualisation" or "specialisation" of the handler. A handler can be seen as a raw opcode, without any operands. It is necessary to decrypt and follow the bytecode to specialise the handler and thus obtain the real semantics of the handler-virtual instruction couple. Getting all the virtual instructions associated with their control flow graph means recovering the original implemented algorithm in clear.

Now, how can we specialise a handler? Using the symbolic context of the host processor! To solve or reduce the various expressions that compose the binding, one can simply inject the state of the symbolic context before the execution into the computed binding. We already have at our disposal methods allowing us to solve memory indirections referring to program data, which is very useful for solving expressions using known memory pointers. The symbolic execution of a handler finally amounts to the application of its solved (or specialised) binding to the context (the context is updated); the symbolic execution of the program is done by the symbolic execution of each of its virtual instructions.

In order to get a better view of this process, let's look the next example, in which every step is detailed:

1. Handler is disassembled.
2. Code is optimised (Fig. 13).
3. Raw binding is computed (Fig. 14).
4. Current context is acquired (Fig. 12).
5. The binding is specialised using symbolic execution (Fig. 15).
6. Context is updated (Fig. 16).

Steps 1 and 2 are totally automated. We have already seen their internal workings. If the handler description is not in the cache, it is computed when needed, on demand.

The current context before the execution of the virtual instruction can be seen in Fig. 12.

```
dword ptr [esp] := edi+4*((((byte ptr[esi]+-(ebx&0ffh))&0ffh)-63h)&0ffh)
eax := (edi+4*((((byte ptr [esi]+-(ebx&0ffh))&0ffh)-63h)&0ffh))
ebx := (ebx&0ffffff00h)|(((ebx&0ffh)+((((byte ptr[esi]+ -(ebx&0ffh))
&0ffh)-63h)&0ffh))&0ffh)
esi := (esi+1)&0ffffffffh
esp := (esp-4)&0ffffffffh
ip := 1000da0eh
```

**Fig. 14** Raw binding

```
dword ptr [esp] := edi+1ch
eax := edi+1ch
ebx := 1001674ah
esi := 10016717h
esp := esp-4
ip := 1000da0eh
```

**Fig. 15** Solved binding

```
dword ptr [esp] := edi+1ch
eax := edi+1ch
ebp := ebp
ebx := 1001674ah
ecx := 1
edx := edx
esi := 10016717h
esp := esp-4
ip := 1000da0eh
```

**Fig. 16** Context after symbolic execution

```
lea eax, dword ptr [edi+1C]
push eax
```

**Fig. 17** Generated native assembly

The disassembled and optimised code can be seen in Fig. 13. The control stub, computing the next handler's address is not displayed for clarity.

Figure 14 exhibits the raw binding. It is easy to see that all the semantic elements are present:

- the write access on the top of the stack,
- incrementation of the `esi` register due to the `lodsb` instruction
- etc.

The context is then injected into the raw binding and the symbolic expressions are resolved (Fig. 15). In this example, we discover that the handler pushes the symbolic value `edi+1C` on top of the stack.

Finally, writings done by the handler are passed to the symbolic context. Figure 16 exhibits the updated context after the symbolic execution of the handler.

Using symbolic execution, we are now able to:

- compute the next handler index;
- compute the next virtual instruction address;
- and thus we can decrypt the whole bytecode according to the control flow of the virtual machine.

### 2.5 Back to the roots

By using all of our previous results, we were able to easily generate the native assembly code corresponding to the specialised handler (Fig. 17) (actually, it is the one we studied during the symbolic execution presentation). These few lines of assembly are the textual representation of a virtual opcode, totally decrypted and specialised into its control flow.

Implicitly, we have taken a crucial step by generating this native assembly. Using the specialised binding of the handler, we have compiled a virtual instruction into its equivalent in native `Ia32` assembly. We do this for all of the virtual instructions, taking into consideration jumps and labels. Then, considering that `Metasm` has a built-in compiler, we can automatically generate the corresponding native machine code. This result is a direct application of second Futamura's projection previously cited. We have created a kind of $L_{bytecode} \rightarrow L_{Ia32}$ compiler. To be more precise, we have specialised the interpreter. Like every compiler, we will now optimise the generated code.

### 2.6 Compilation: the summer's hit

The previous step gives us the bytecode of the virtual machine compiled into native `x86` machine code. This usage of compilation techniques has to be put into perspective with works proposed by Rolles [14] targeting the defeat of virtual machine like `VMProtect` using compilation. In his approach, virtual machine bytecode was first translated into an intermediate representation to be optimised and then compiled. We made the choice to directly translate bytecode into assembly, using code symbolic semantics and specialised code from the handler.

The compilation of the assembly is not a problem. Nevertheless, given the huge number of stack instructions (`push-pop`), it clearly appears the virtual machine behaves like kind of a stack automaton (Fig. 18). This aspect of the code is a problem, as it complicates the understanding of the code and is the source of an important overhead.

Once again, we will use our optimisation engine. It will automatically clean this stack-based aspect (Fig. 19) without requiring additional work.

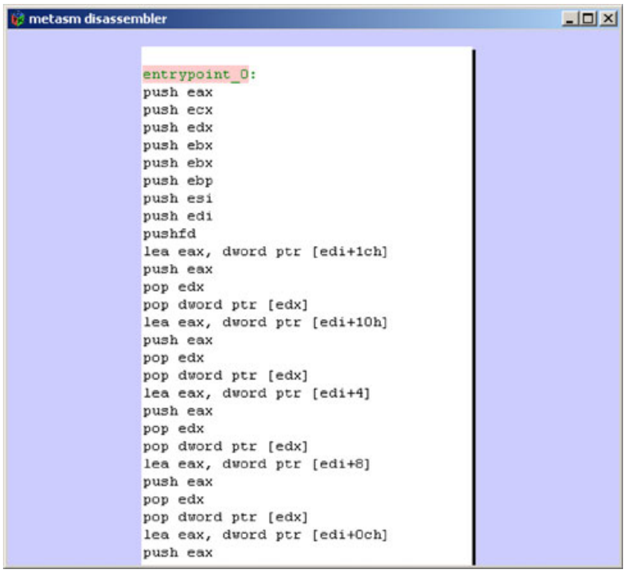


**Fig. 18** Entry point of *un-virtualized* code

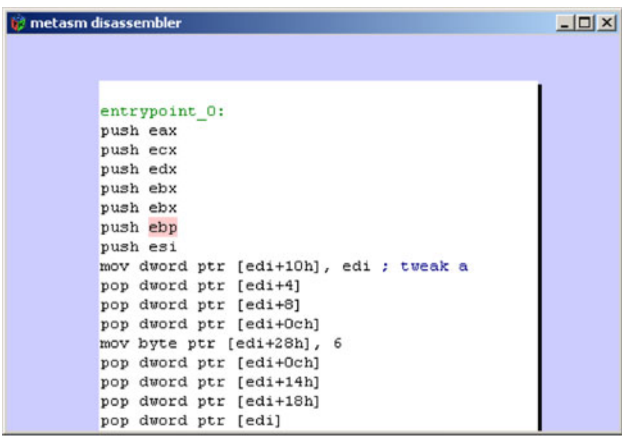


**Fig. 19** Entry point of *un-virtualized* slightly optimised

## 2.7 Everybody's gone surfing

Even if most of the previous steps are automated and globally generic, the next one requires some manual analysis. The code displayed in Figs. 18 and 19 is an introduction to this next step.

This code is present throughout all the virtualized functions prologues. All the host processor registers are pushed on the stack and then popped in memory areas, referred to using `dword ptr [edi+xx]`-like indirections. These indirections simply refer to the virtual context of the virtual machine. To summarise, native registers are directly mapped onto virtual registers. The inverse process is performed in each function epilogue. This analysis, while precious, is specific to the target, thus there is a loss of genericity. Nevertheless it is important to point out that in the final code, all the virtual machine artifact code will be removed.

Now that we are aware of the virtual context, the problem can be expressed in these words: how can we abstract the virtual machine registers in our `Ia32` disassembler?

The answer is actually quite simple: it is enough to extend the `Ia32` processor that is used by `Metasm` to add these virtual registers. The `Ruby` code use to create this extension is given in Fig. 20 for information. This is the key quality of a framework like `Metasm`: each part is easily scriptable and can be adapted to several usages, even the most obscure ones. From now on, an "extended" virtual register will be seen and manipulated exactly like a native register.

Once the processor is extended, we walk through the instruction flow to *inject* the new registers on-the-fly; they will replace the inexpressive and complex indirections. Moreover, all of our optimisation methods can now be silently applied to virtual registers. This last step complete the defeat of the virtual code. Let's see in practice the result of the optimisation engine on a small code section, step by step:

1. Original code, generated by the compiler's $L_{bytecode} \rightarrow L_{Ia32}$:

```
71h push 1000a2b4h
76h pop edx
77h mov eax, dword ptr [edi+2ch]
7ah add edx, dword ptr [edi+2ch]
7dh push dword ptr [edx]
7fh mov eax, dword ptr [esp]
82h pop ecx
83h xor dword ptr [esp], eax
```

**Fig. 20** `Ia32` processor extension

```
def extend_ia32cpu

  Ia32::Reg.i_to_s[32].concat( %w[virt_eax virt_ecx])
  Ia32::Reg.s_to_i.clear
  Ia32::Reg.i_to_s.each { |sz, hh|
     hh.each_with_index { |r, i|
        Ia32::Reg.s_to_i[r] = [i, sz]
     }
   }
   Ia32::Reg::Sym.replace Ia32::Reg.i_to_s[32].map { |s| s.to_sym }

end
```

```
86h push\-fd
87h pop dword ptr [edi+1ch]
8ah lea eax, dword ptr [edi+18h]
8dh push eax
8eh pop edx
8fh pop dword ptr [edx]
91h lea eax, dword ptr [edi+18h]
94h push eax
95h lea eax, dword ptr [edi]
97h push eax
98h pop edx
99h pop edx
9ah push dword ptr [edx]
9ch push 1000a2d4h
0a1h pop edx
0a2h mov eax, dword ptr [edi+2ch]
0a5h add edx, dword ptr [edi+2ch]
0a8h pop dword ptr [edx]
```

2. After the fist step: virtual registers have been injected (Cf. `virt_ecx` or `virt_eax`) in the code that has been manipulated. Some virtualization issues like computation from relative address to absolute address have also been wiped (references to `dword ptr [edi+2ch]`). The number of stack operations is also reduced.

```
61h push dword ptr [1000a2d4h]
63h lea eax, virt_ecx
68h pop dword ptr [eax]
6ah lea eax, virt_ecx
6fh push dword ptr [eax]
7dh push dword ptr [1000a2b4h]
82h pop ecx
83h xor dword ptr [esp], ecx
8ah lea eax, virt_ecx
8fh pop dword ptr [eax]
91h lea eax, virt_ecx
94h push eax
95h lea eax, virt_eax
99h pop edx
9ah push dword ptr [edx]
0a8h pop dword ptr [1000a2d4h]
```

3. The number of stack operations (computations or simple movements) is, once again, reduced.

```
61h push dword ptr [1000a2d4h]
68h pop virt_ecx
6fh push virt_ecx
82h mov ecx, dword ptr [1000a2b4h]
83h xor dword ptr [esp], ecx
8fh pop virt_ecx
91h lea eax, virt_ecx
99h mov edx, eax
9ah push dword ptr [edx]
0a8h pop dword ptr [1000a2d4h
```

4. We progressively rebuild the initial data movements. For the moment at least, two read access clearly appears (at address 68h and 83h).

```
68h mov virt_ecx, dword ptr [1000a2d4h]
83h xor virt_ecx, dword ptr [1000a2b4h]
91h lea eax, virt_ecx
9ah push dword ptr [eax]
0a8h pop dword ptr [1000a2d4h]
```

5. The last `lea` (*Load effective address*) instruction is now reduced. We have a very concise code.

```
68h mov virt_ecx, dword ptr [1000a2d4h]
83h xor virt_ecx, dword ptr [1000a2b4h]
0a8h mov dword ptr [1000a2d4h],
    virt_ecx
```

In this brief example, the 26 lines of original code have been reduced to only 3 lines of code. Useless movements or computations have been wiped. The code can now be easily understood. This example is quite representative of the effectiveness of the whole optimisation process. Finally, in the unprotected code, we found some distinctive sequences shared by some code which has not been virtualized: we can thus say that our work is done here. This result is quite satisfying.

### 2.8 Partial conclusion

Here we have presented a concrete approach which uses the binary manipulation framework `Metasm`. The concepts we have used rely on strong theoretical results: partial evaluation and specialisation [15]. The key idea of specialisation is to delete all interpreted elements. During our analysis, we took advantage of all static information which allowed us to compute partial results: computation of arithmetical operations used by the obfuscation process, computation of

the result of the application of the interpreter to the bytecode. Finally, once the rewriting system has been applied to the code, we have a specialised program, according to our own performance criteria, namely code conciseness and ease of analysis. Moreover, this specialisation is almost optimal: we have suppressed the interpreted code completely and recovered code that is highly similar to the unprotected original [16].

More than a raw technical demonstration, this result is also an overview of a possible new requirement for security products analysis engine like anti-virus software or automatic classification tools. As noted previously, packers are more or less well handled, but virtualization is still an issue. A tool like the one we have developed makes it possible to recover a code which is equivalent to an unprotected binary, automatically, for every instance of the protection. It is then quite easy to detect variants from the same strain for example. Nevertheless, it requires some heavy computation, such analysis may thus be delegated to dedicated servers with sufficient resources (hardware and time), or to *cloud computing* technologies (which is the same in the end).

Our results are definitively positive, in particular for code deobfuscation; however, we have to put them back in their context. Indeed, for example, our optimisation engine only works locally inside basic blocks, it may be relatively easy to circumvent it by adding extra-procedural obfuscation. A lack of a good intermediate representation clearly appears, we need to be able to handle some higher level concepts like loops, functions, etc. In the same vein as our decompilation work from last year, we have chosen decompilation as a new work axis in support of our optimisation engine. But we will now have to deal with `C` code generation problems.

## 3 Decompilation

During the works presented previously, a big step was the reconstruction of native assembly code from a *binding*, i.e. a bunch of affectations.

This is a tedious phase, because the assembly language, beside its being tied to a fixed architecture, imposes severe constraints. For instance, in `x86`, it is impossible to have two memory addresses referenced by a single mnemonic. This forces us to know and work around those limitations, maybe by generating two instructions to achieve a single operation. We also need to know precisely which instruction is used to add two registers, or one register with a numeric constant …

On the other hand manipulating `C` code is much easier. In fact, the binding is a set of affectations, which can be directly written in `C`.[2] The `C` representation also totally hides the `CPU` flags handling, which all the more lightens the data that have to be handle. Finally, using `C` simplifies the control graph handling: it is in fact simpler to walk an *if/else* node than to interpret and follow an assembly conditional jump.

Of course, all this needs a correct translation of assembly code to `C` code, which may be very challenging to achieve.

### 3.1 Decompilation mechanisms

The decompilation module inside the framework is a work in progress. Here we'll present the internal workings as they are now, but be wary that later versions may differ significantly.

From a given entrypoint in the disassembly graph, the graph is walked to find basic block dependency, with respect to their register writing and reading (*producer/consumer*). Each block is annotated with the list of registers that are needed by another block later in a codepath, without having been rewritten in between.

Every block then has its binding computed. In it, accesses to registers and/or memory which backtrace to a stack offset are replaced by the symbolic value of the function frame pointer, which shows the use of local variables as typically done by compilers.

A correct binding depends on an accurate emulation of all the instructions of the block. If one of those instructions is not perfectly supported, it is shown as is in the `C` listing, using an *inline assembly* construction. This will generally indicate incorrect code (due to the registry dependencies), and will certainly be fatal for the automatic refinement to come.

From the binding, the dependency sequence of a block is turned into a `C` expression sequence. This operation must be performed carefully: the binding represents a set of simultaneous affectations, which must be rewritten as a sequence of affectations; great care must be taken with inter-expression dependency. This may prompt the use of temporary variables.

Then the last instruction of the block is inspected to discover the mechanism that should be used to translate the original control flow transition: subfunction call, use of a *goto* …

In the case of the subfunction call, the decompilation process is recursive, so at this point we should already know the subfunction prototype and ABI.[3] The value used for the arguments are formatted in accordance with the `C` standard. Conditional jumps are translated to *if () goto*.

All of those `C` expressions are stored in the body of a `C` function, which holds the translation of all the code that is accessible from the chosen entrypoint.

This limitation is needed (the `C` standard forbids expressions in the global scope), but it may also be misleading: a `C` function is supposed to have a formatted behaviour (stack

[2] We must however take care of variable dependency, as the binding represents a set of simultaneous affectations.

[3] Application Binary Interface: describes how arguments are passed to the subfunction and the like.

pointer conservation…) that may not correspond to the underlying assembly code. Further work will be needed to detect those occurrences and show the discrepancies (maybe through the use of nonstandard C attributes).

From now on, we won't need to manipulate assembly code anymore. But much work is still needed; most notably C control structure reconstruction and variable type recovery.

Here is what the code looks like at this point:

```
int sub_48h()
{
    register int eax;
    register int frameptr;
sub_48h:
    *(__int8*)(frameptr-12) = 0;
    *(__int32*)(frameptr-16) = *(__int32*)(
        frameptr+4);
    eax = 8;

loc_57h:
    eax = eax-1;
    if (eax == 0)
        goto loc_124h;
    *(__int8*)(frameptr-12) = *(__int8*)(*(
        __int32*)(frameptr-16));
    sub_244h(frameptr-16);
    goto loc_57h;

loc_124h:
    return eax;
}
```

The next step is to simplify the control graph, for example by fixing a useless goto (e.g. a goto pointing to another goto). Then the code is parsed to try to identify structures that may be translated to standard *if* and *while*. Unused labels are removed.

Most of the time, when working with a C compiler-generated code, gotos are no longer visible, and we can see the code structure quite clearly.

Variable types are then inferred from existing code. The principal source for the types is the prototype of the subfunctions which is a strong indication for the types of the expressions used as arguments. Indirections also give us clues on the more basic types (integers, pointers). Affectations are used to propagate direct and indirect (pointed …) types.

This pass may generate conflicting types for a stack offset; in this case one of the types is chosen and C casts are used where needed. An antialiasing algorithm is in development, which should solve this kind of problems.

The aliasing problem is prevalent with the registers, because they see a large number of unrelated affectations throughout the function, and pollute the types of related variables. Use of the *union* construct in the original code also leads to the same kinds of problems.

The antialiasing algorithm will do a *liveness* analysis of variables, and for each domain found a specific variable may be forked from the existing one. Each clone has its own (correct) type found through the method explained previously. However the code is really immature at the time of this writing.

The current prototype is somewhat *x86* biased, but gives promising results. Still, more work is needed before we can use it for code deobfuscation.

### 3.2 Use

From now on, we will postulate that we have a full functional decompiler.

It would allow us to use some of the existing tools that provide code optimisation functionalities (LLVM[4] for example). There is still one problem left: we have a very particular optimisation criteria. It may be expressed as *code understandability*. In our point of view, speed of execution, for example, is a side effect. Thus, it is possible that many legacy optimisation algorithms actually lead to a more complex form of the code, that will be more difficult to understand.

It is very likely that the same techniques we have used at the assembly level will work fine at a higher level; furthermore optimisation could even be done with an intra-procedural or inter-procedural scope.

If we manage to integrate the previously discussed contextualisation mechanism into the decompilation stage, the bytecode of some virtual machines (like the one studied in the first part of this paper) may directly be decompiled. This prospect seems promising and will be investigated in the future.

Taking into consideration the great simplicity of the virtual machine's handlers, it may also be possible to directly emulate some parts of them once translated in C, and then to proceed to the contextualisation step.

Currently, these are only perspectives but they seem quite realistic.

## 4 Pastoral concolic execution

As a conclusion to this paper, we would like to introduce another possible extension of our works. In this part, we will not focus on the result at all, but on the analysis approach and the use of a Metasm. Here is the context of this study: we analyse a protected binary, we know that the many virtual machines are used but we are not able to get or to simulate their whole initialisation (their context is too large, the code is heavily obfuscated, etc.). Still, how can we proceed in order to get a discerning view of the protection workings?

The approach we propose here is said to be concolic. This adjective, commonly used in software testing, points out an approach which couples a real execution with a symbolic execution of a program. That is exactly what we will do,

4 http://llvm.org/.

**Fig. 21** Mixing the debugger with the static analysis

```
1  def debugloop
2     debugevent = debugevent_alloc
3        while not @mem.empty?
4           if WinAPI.waitfordebugevent(debugevent, 500)
5              debugloop_step(debugevent)
6           else
7              load 'starter.rb'
8           end
9        end
10    end
11 end
12
13 def handler_newthread(pid, tid, info)
14    super
15    puts "Setting break on vm entry\n\n"
16    set_hbp(@vm_entry, pid, tid)
17    WinAPI::DBG_CONTINUE
18 end
19
20 def handler_exception(pid, tid, info)
21    case info.code
22       when Metasm::WinAPI::STATUS_SINGLE_STEP
23          case get_context(pid, tid)[:eip]
24          when @vm_entry
25             puts "\n######   BREAK ON VM ENTRY   ######\n"
26
27             ctx = get_context(pid, tid)
28             remote_mem = OS.current.findprocess(pid).mem
29
30             sa = Static_analyzer.new(remote_mem, ctx[:esp])
31             sa.followHandlers
32
33             update_eip()
34          end
35       end
36    super
37    WinAPI::DBG_CONTINUE
38 end
```

using a part of dynamic analysis (thanks to a debugger) and a part of static analysis (as we have already largely discussed in the paper).

## 4.1 Dynamic analysis

Metasm proposes a wrapper (very basic in its current version) on the standard Microsoft Windows debug API. We are interested by the analysis of the virtual machine, logically we will break at the entry point of this virtual machine. Then we will proceed to some static analysis.

A code skeleton is included in Fig. 21. A few functions like `update_eip`, are not detailed here for the sake of simplicity and conciseness, but they are very basic and typical.

The fundamental component is located in lines 26 and 27. The variable `ctx` actually is the context of the debugged process, we can then access the value of each of the registers of the current thread. The variable `remote_mem` allow us to get access to the whole process memory. These two elements will be injected into the static analysis to increase its discernment.

## 4.2 Definition of a virtual processor

A quick and dirty dynamic analysis of the virtual machine reveals many interesting elements: different keys are used to decrypt the opcodes, there is a set of flags, etc. We know where these elements are located in memory and how they are accessed. Actually, we are able to describe the symbolic binding of the virtual machine (Fig. 22).

As an example, the first line of the hash structure means that at the location pointed by `dword ptr [esp+10h]`, we find the symbol `key_a`. We then define the memory mapping of each of the relevant symbols.

## 4.3 Simplified symbolic execution

We now want to proceed to the symbolic execution of the bytecode on the virtual processor. Keep in mind that we made the hypothesis that we were not able to precisely analyse the virtual machine initialisation. How can we initialise the context of the virtual processor?

Actually, there is no need for that. We already have all the information required in the context and memory of the debugged process: we only need to read this memory. We have developed a small method that fits in the hand (Fig. 23).

A result of a call to `vm_ctx_init` can be seen in Fig. 24.

A numeric value has been associated with each symbol. The `solve_ind_partial` method has already been presented. It was initially able to solve indirections pointing to the program's memory (present in its code or data sections). We have extended its functionality to consider the whole

**Fig. 22** Symbolic binding definition

```
#------------------------   VM symbolic bindings ------------------------#
@symbolic_binding = {
  Indirection[Expression[:esp, :+, 0x10], 4, nil] =>  Expression[:key_a],
  Indirection[Expression[:esp, :+, 0x14], 4, nil] =>  Expression[:key_b],
  Indirection[Expression[:esp, :+, 0x18], 4, nil] =>  Expression[:key_c],

  Indirection[Expression[:esp, :+, 0x58], 4, nil] =>  Expression[:delta],
  Indirection[Expression[:esp, :+, 0x5c], 4, nil] =>  Expression[:delta_false],
  Indirection[Expression[:esp, :+, 0x60], 4, nil] =>  Expression[:delta_true],

  Indirection[Expression[:esp, :+, 0x134], 1, nil] =>  Expression[:flag8],
  Indirection[Expression[:esp, :+, 0x135], 1, nil] =>  Expression[:flag7],
  Indirection[Expression[:esp, :+, 0x136], 1, nil] =>  Expression[:flag6],
  Indirection[Expression[:esp, :+, 0x137], 1, nil] =>  Expression[:flag5],
  Indirection[Expression[:esp, :+, 0x138], 1, nil] =>  Expression[:flag4],
  Indirection[Expression[:esp, :+, 0x139], 1, nil] =>  Expression[:flag3],
  Indirection[Expression[:esp, :+, 0x13a], 1, nil] =>  Expression[:flag2],
  Indirection[Expression[:esp, :+, 0x13b], 1, nil] =>  Expression[:flag1],

  Indirection[Expression[:esp, :+, 0x13c], 4, nil] => Expression[:nrHandler],
}
#------------------------   VM symbolic bindings ------------------------#
```

**Fig. 23** Virtual processor automatic initialization

```
def vm_ctx_init()
   vmctx = {}
   @symbolic_binding.each_value{ |key|
      vmctx[key.reduce_rec] = solve_ind_partial(
            @symbolic_binding.dup.invert[Expression[key.reduce_rec]],
            true
         )
   }
   vmctx
end
```

```
delta := 250210h
delta_false := 0
delta_true := 25d568h
flag1 := 74h
flag2 := 35h
flag3 := 43h
flag4 := 7eh
flag5 := 60h
flag6 := 5fh
flag7 := 25h
flag8 := 0
key_a := 110h
key_b := 2
key_c := 2595a8h
nHandler := 0ce3h
```

**Fig. 24** Fully initialized virtual context

process memory. Using the virtual context, we are now able to proceed to the symbolic execution of the bytecode.

## 4.4 Symbolism injection

Here is the code of a virtual machine handler (Fig. 25).

We compute the binding of this piece of code. Keep in mind that Metasm kindly offers a method for this to be performed automatically. In this particular case, we are only interested in the memory writings. The ugly result is exhibited in Fig. 26.

Well, it still is not easy to understand what happens here. But wait, in this listing there are many elements we know. For example, we know that the symbol `key_a` is hiding behind the indirection `dword ptr [esp+10h]`.

Once again, the solution is quite trivial: we have to inject the symbolic binding into the handler's binding. In `Ruby`, we will simply write:

```
expression.bind(@symbolic_binding)
```

We then get the following intermediate result (Fig. 27).

This result is encouraging, we now have to apply the `solve_ind_partial` method to each of the intermediate expressions. Here is an extract from the verbose log of this method (Fig. 28):

The symbolic elements are progressively injected. Indirections are solved, sometime until a numerical value is obtained. As a side effect, *memory aliasing* (several pointers

**Fig. 25** A handler's code

```
412eb3h mov esi, dword ptr [esp+14h]
412eb7h mov ecx, dword ptr [esp+18h]
412ebbh mov ebx, dword ptr [esp+10h]
412ebfh mov eax, dword ptr [436000h+4*esi]
412ec6h mov edi, dword ptr [436000h+4*ecx]
412ecdh mov edx, dword ptr [436000h+4*ebx]
412ed4h mov ebp, dword ptr [436004h+4*ecx]
412edbh xor eax, edi
412eddh mov edi, dword ptr [esp+10h]
412ee1h xor eax, edx
412ee3h mov ebx, dword ptr [esp+140h+4*eax]
412eeah mov eax, dword ptr [436004h+4*esi]
412ef1h mov edx, dword ptr [436004h+4*edi]
412ef8h mov esi, dword ptr [esp+14h]
412efch xor eax, ebp
412efeh xor eax, edx
412f00h mov ebp, dword ptr [esp+10h]
412f04h mov edx, dword ptr [esp+140h+4*eax]
412f0bh mov eax, dword ptr [436008h+4*esi]
412f12h mov esi, dword ptr [esp+18h]
412f16h mov edi, dword ptr [436008h+4*ebp]
412f1dh mov ecx, dword ptr [436008h+4*esi]
412f24h xor eax, ecx
412f26h xor eax, edi
412f28h mov ecx, dword ptr [esp+140h+4*eax]
412f2fh mov edi, dword ptr [esp+10h]
412f33h movzx eax, byte ptr [edx]
412f36h mov edx, dword ptr [43600ch+4*edi]
412f3dh test al, al
412f3fh mov byte ptr [ebx], al
412f41h mov eax, dword ptr [esp+14h]
412f45h setz byte ptr [ecx]
412f48h mov ecx, dword ptr [43600ch+4*esi]
412f4fh add edi, 4
412f52h mov dword ptr [esp+10h], edi
412f56h mov ebp, dword ptr [43600ch+4*eax]
412f5dh add esi, 4
412f60h mov dword ptr [esp+18h], esi
412f64h add eax, 4
412f67h mov dword ptr [esp+14h], eax
412f6bh xor ecx, ebp
412f6dh xor ecx, edx
412f6fh mov dword ptr [esp+13ch], ecx
412f76h jmp loc_401f20h
```

**Fig. 26** Handler's raw binding

```
byte ptr [dword ptr [esp+4*(dword ptr [4*dword ptr [esp+14h]+436000h]^
(dword ptr [4*dword ptr [esp+18h]+436000h] ^dword ptr [4*dword
ptr [esp+10h]+436000h]))+140h]] := (byte ptr [dword ptr [esp+4*(dword ptr
[4*dword ptr [esp+14h]+436004h]^(dword ptr [4*dword ptr [esp+18h]+436004h]^dword
ptr [4*dword ptr [esp+10h]+436004h]))+140h]]&0ffh)

byte ptr [dword ptr [esp+4*(dword ptr [4*dword ptr [esp+14h]+436008h]^(dword ptr
[4*dword ptr [esp+18h]+436008h]^dword ptr [4*dword ptr
[esp+10h]+436008h]))+140h]] := ((byte ptr [dword ptr [esp+4*(dword ptr [4*dword
ptr [esp+14h]+436004h]^(dword ptr [4*dword ptr [esp+18h]+436004h]^dword ptr
[4*dword ptr [esp+10h]+436004h]))+140h]]&0ffh)==0)

dword ptr [esp+13ch] := (dword ptr [4*dword ptr [esp+18h]+43600ch]^(dword ptr
[4*dword ptr [esp+14h]+43600ch]^dword ptr [4*dword ptr [esp+10h]+43600ch]))

dword ptr [esp+10h] := dword ptr [esp+10h]+4
dword ptr [esp+14h] := dword ptr [esp+14h]+4
dword ptr [esp+18h] := dword ptr [esp+18h]+4
```

on the same memory area) is de facto defeated. Numerical values (addresses) are converted into their symbolic equivalents, when possible. The final result (Fig. 29) is quite clear and understandable.

### 4.5 The beauty of gesture

In this section, we wanted to illustrate a concolic approach on a virtual machine-based software protection. Results are quite encouraging and sometimes offer amazing shortcuts. We make use of the information from both code and runtime memory. We liberate ourselves from some constraints and limitations due to pure static analysis. Doing so, we greatly simplify virtual machine handler analysis. A key issue is to decide when to solve expression to its numerical value and when to stay in symbolic representation. In this example, we are able to hook every call to the virtual machine, to proceed to the symbolic execution of the bytecode, to update both

```
byte ptr [dword ptr [esp+4*(dword ptr [4*key_b+436000h]^(dword
ptr [4*key_c+436000h]^dword ptr [4*key_a+436000h]))+140h]] := byte ptr [dword
ptr [esp+4*(dword ptr [4*key_b+436004h]^(dword ptr [4*key_c+436004h]^dword
ptr [4*key_a+436004h]))+140h]]&0ffh

byte ptr [dword ptr [esp+4*(dword ptr [4*key_b+436008h]^(dword
ptr [4*key_c+436008h]^dword ptr [4*key_a+436008h]))+140h]] := (byte ptr
[dword ptr [esp+4*(dword ptr [4*key_b+436004h]^(dword ptr
[4*key_c+436004h]^dword ptr [4*key_a+436004h]))+140h]]&0ffh)==0

key_a := key_a+4
key_b := key_b+4
key_c := key_c+4

nHandler := dword ptr [4*key_c+43600ch]^(dword ptr [4*key_b+43600ch]^dword
ptr [4*key_a+43600ch])
```

**Fig. 27** Intermediate binding

```
 - solve read access to arg: (dword ptr [4371ech]^(dword ptr [437888h]^dword ptr
[43b780h]))&0ffffffffh
 - solved key: 184dh

 - solve write access to arg: byte ptr [dword ptr [esp+4*(dword ptr
[4*key_b+436000h]^(dword ptr [4*key_c+436000h]^dword ptr
[4*key_a+436000h]))+140h]]
 - make stack variable <esp+136h> from stack address 23e9a6h
 - solved key: flag6

 - solve read access to arg: byte ptr [dword ptr [esp+4*(dword ptr
[437880h]^(dword ptr [4371e4h]^dword ptr [43b778h]))+140h]]&0ffh
 - solved key: flag5&0ffh

 - solve write access to arg: byte ptr [dword ptr [esp+4*(dword ptr
[4*key_b+436008h]^(dword ptr [4*key_c+436008h]^dword ptr
[4*key_a+436008h]))+140h]]
 - make stack variable <esp+135h> from stack address 23e9a5h
 - solved key: flag7

 - solve read access to arg: ((byte ptr [dword ptr [esp+4*(dword ptr
[437880h]^(dword ptr [4371e4h]^dword ptr [43b778h]))+140h]]&0ffh)==0)&0ffffffffh
 - solved key: ((flag5&0ffh)==0)&0ffffffffh
```

**Fig. 28** `solve_ind_partial` method trace

```
flag6 := flag5&0ffh
flag7 := ((flag5&0ffh)==0)&0ffffffffh
key_a := 15e1h
key_b := 623h
key_c := 47ch
nHandler := 184dh
```

**Fig. 29** Final binding

the context and the memory of the process and then return to the original code. Once again we are close to compilation concepts. What we do is a kind of just-in-time compilation of the bytecode to our own interpreter.

## 5 Conclusion

Compared to our previous works which we presented last year [1], we have stepped further into the automation of obfuscating and virtual machine based protections code analysis. Optimisations used in the first part of this paper are generic and quite simple. Our implementation is really basic and we miss a strong intermediate representation that would be able to support higher level optimisations (like REIL [17] for example). We have been working towards decompilation. It should allow us to reach a greater level of genericness.

Another step has also been taken by using the semantics of instructions. The use of bindings once again reveals itself to be quite powerful. We have defeated a virtual machine without even analysing its handlers. Still, the extraction of the interpreter's semantics allowed us to generate a compiler from bytecode to native x86 assembly. This kind of approach may have been used as a preparatory phase for a malicious code detection engine [11].

We should also note that our work relies on the hypothesis that we are able to disassemble most of the code we study. Some techniques, like memory aliasing, may try to exploit current limitations of the backtracking engine and emulation abilities of the framework to disrupt the recovery of the control flow.

In order to be resilient to such kinds of threats, the third part (dealing with a concolic approach) is very promising. It allows the analyst to get complete control over all the

elements. This ability, combined with the manipulation of symbolic elements, leads to more fun and powerful code analysis sessions.

## References


1. Guillot, Y., Gazet, A.: Semi-automatic binary protection tampering. J. Comput. Virol. **5**(2), 119–150 (2009)
2. Guillot, Y.: Metasm. In: 5ème Symposium sur la Sécurité des Technologies de l'Information et des Communicatins (SSTIC'07). http://actes.sstic.org (2007)
3. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**, 121–189 (1995)
4. Wroblewski, G.: General method of program code obfuscation (2002)
5. Beck, J., Eichmann, D.: Program and interface slicing for reverse engineering. In: In IEEE/ACM 15th Conference on Software Engineering (ICSE'93), pp. 509–518. IEEE Computer Society Press (1993)
6. Quist, D., Valsmith: Covert debugging—circumventing software armoring techniques (2007)
7. Bohne, L.: Pandora's bochs: automated malware unpacking (2008)
8. Kang, M.G., Poosankam, P., Yin, H.: Renovo: a hidden code extractor for packed executables (2007)
9. Perriot, F.: Defeating polymorphism through code optimization (2003)
10. Webster, M., Malcolm, G.: Detection of metamorphic and virtualization-based malware using algebraic specification. In: EICAR (2008)
11. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H.: Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA (Nov 2005)
12. http://orange-bat.com
13. Futamura, Y.: Partial evaluation of computation process—an approach to a compiler-compiler. Syst. Comput. Controls **2**, 45–50 (1971)
14. Rolles, R.: Optimizing and compiling (2008)
15. Marlet, R.: Vers une formalisation de l'évaluation partielle. PhD thesis, L'Université de Nice - Sophia Antipolis, École Doctorale - Sciences pour l'Ingénieur (1994)
16. Hartmann, L., Jones, N.D., Simonsen, J.G.: Interpretive overhead and optimal specialisation
17. Dullien, T., Porst, S.: Reil: a platform-independent intermediate representation of disassembled code for static code analysis. In: CanSecWest (2009)